DEDUKTI: A Universal Proof Checker

Mathieu Boespflug Quentin Carbonneaux Gilles Dowek
Olivier Hermant Ronan Saillard

Deducteam INRIA

MINES ParisTech

January 9, 2013

What is Dedukti?

What is Dedukti?

- A type-checker for the $\lambda\Pi$-calculus modulo
- A theory-independent proof-checker
- With CoqInE a tool to recheck proofs from Coq
- With HOLidE a tool to recheck proofs from HOL/OpenTheory
- A logical framework with rewrite rules
- A framework to study interoperability
- A type-checker with new implementation techniques

Table of Contents

The $\lambda\Pi$-calculus modulo

Dedukti Core

- type theory: the $\lambda\Pi$-calculus (dependent types):

$$\mathit{array} \;:\; \mathit{nat} \to \mathrm{Type}$$

- enriched with rewrite rules

$$\mathit{head}\ (\mathit{hd} :: \mathit{tl}) \quad \longrightarrow \quad \mathit{hd}$$

- used to generalize the conversion rule
- can encode all the Functional PTSes [Cousineau & Dowek, 2007]

Typing Rules

$$s \in \{\text{Type}, \text{Kind}\}$$

$$(\mathit{sort})\ \frac{\Gamma\ \text{Well-Formed}}{\Gamma \vdash \text{Type} : \text{Kind}} \qquad (\mathit{var})\ \frac{\Gamma\ \text{Well-Formed} \qquad x{:}A \in \Gamma}{\Gamma \vdash x : A}$$

$$(\mathit{prod})\ \frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x{:}A \vdash B : s}{\Gamma \vdash \Pi x{:}A.\ B : s}$$

$$(\mathit{abs})\ \frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x{:}A \vdash B : s \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\ M : \Pi x{:}A.\ B}$$

$$(\mathit{app})\ \frac{\Gamma \vdash M : \Pi x{:}A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : \{N/x\}B}$$

$$(\mathit{conv})\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ A \equiv_{\beta\mathcal{R}} B$$

Figure: Typing rules for the $\lambda\Pi$-calculus modulo

An Example

An example

example.dk

```
Nat : Type.
Z   : Nat.
S   : Nat -> Nat.
two : Nat := S (S Z).

Bool : Type.
true : Bool.
false: Bool.
prf  : Bool -> Type.
tt   : prf true.

eq : Nat -> Nat -> Bool.
[ ] eq Z Z --> true
[n:Nat, m:Nat] eq (S n) (S m) --> eq n m
[n:Nat] eq (S n) Z --> false
[n:Nat] eq Z (S n) --> false.

List : n:Nat -> Type.
Nil  : List Z.
Cons : n:Nat -> x:Nat -> List n -> List (S n).

head : n:Nat -> List (S n) -> Nat.
[n:Nat, x:Nat, tl:List n] head n (Cons {n} x tl) --> x.

theorem0: prf (eq (head Z (Cons Z two Nil)) two) := tt.
```

Implementation

The big picture

Dedukti: goals and weapons

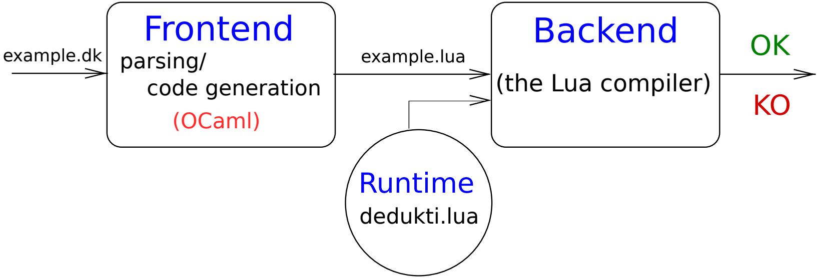
Goals:

- Fast type checking.
- Versatility: efficient for any (embedded) logic.

Philosophy, be lazy:

- Do not reimplement long-standing features.
- Reuse existing tools.

The Big Picture



- Dedukti is a proof-checker **generator**

Advantages

Use of the source language capability:

- **Higher Order Abstract Syntax** (HOAS).
- **Normalisation by Evaluation** (NbE).

Dedukti	Lua
$(\lambda x.M)N$	(function (x) M end)(N)
$head\ (hd :: tl) \longrightarrow hd$ $head\ nil \longrightarrow default$	function head (x) if x.id == Cons and then return x.args[1] elseif x.id = Nil return default end end

Implementation

Type-checking algorithm

Bidirectional/Context-Free Type Checking

Bidirectional type checking:

- Mix of type-checking and type-inference.
- Smaller terms (Curry-style).

Context-Free type checking:

- No search in contexts.
- Type annotation for free variables.

The Resulting System

$\boxed{\vdash M \Rightarrow A}$ the term M synthesizes type A

$$(sort)\ \frac{}{\vdash \text{Type} \Rightarrow \text{Kind}} \qquad (var)\ \frac{}{\vdash [x : A] \Rightarrow A}$$

$$(app)\ \frac{\vdash M \Rightarrow C \qquad C \rightarrow^*_w \Pi x{:}A.\ B \qquad \vdash N \Leftarrow A}{\vdash M\ N : \{N/x\}B}$$

$\boxed{\vdash M \Leftarrow A}$ the terms M verifies type A

$$(abs)\ \frac{C \rightarrow^*_w \Pi x{:}A.\ B \qquad \vdash \{[y : A]/x\}M \Leftarrow \{y/x\}B}{\vdash \lambda x.M \Leftarrow \Pi x{:}A.\ B}$$

$$(prod)\ \frac{\vdash A \Leftarrow \text{Type} \qquad \vdash \{[y : A]/x\}B \Leftarrow s}{\vdash \Pi x{:}A.\ B \Leftarrow s}\ (s \in \{\text{Type}, \text{Kind}\})$$

$$(conv)\ \frac{\vdash N \Rightarrow B \qquad A \equiv B}{\vdash N \Leftarrow A}$$

Figure: An implementation of the $\lambda\Pi$-calculus modulo

Implementation

Versatility

Versatile Type-Checking

The conversion test: how to normalize ?

- Proofs terms with few reductions (ex: from HOL).
  - Best technique: **interpretation** of the λ-terms.
- Proofs terms with a long reduction sequence (ex: proof by reflection (Coq)).
  - Best technique: **compilation** and execution of the λ-terms.

How to choose the correct strategy?

The JIT compromise

- **compile** the computational parts, **interpret** the rest
- **choice delegated** to a cutting edge JIT: luajit

File	Time to process
`fact.hs`	0.7 sec + 0.04 sec
`fact.lua`	0.7 sec
`fact.vo`	3.3 sec
`Coq_Init_Logic.hs`	45 sec + 0.4 sec
`Coq_Init_Logic.lua`	0.4 sec
`Coq_Init_Logic.vo`	0.14 sec

Figure: Compilation vs JIT vs Interpretation

`Coq_Init_Logic` is a module in Coq's prelude, `fact` typechecks the identity with the type $vec\ 8! \rightarrow vec\ 8!$.

Conclusion

Conclusion

Contributions

- Proof-checker generator architecture.
- A algorithm mixing bidirictional and context-free type checking.
- Use of a JIT compiler.

Further work

- Non-linear rewrite rules.
- Check Coq' Standard Library.
- LuaJIT limits.
- More optimisations (convertibility test/normalisation).

Dedukti's website

https://www.rocq.inria.fr/deducteam/Dedukti/

Questions ?

QUESTIONS ?